**COSC 320 – 001**

***Analysis of Algorithms***

2022/2023 Winter Term 2

**Third Milestone**

**Project Topic Number: #2**

**String Matching for Plagiarism Detection**

**Group Lead:**

**Youssef Mahmoud**

**Group Members:**

**Esteban Martínez (22717805),**

**Youssef Mahmoud (37624970),**

**Khalid Mahmoud (28842458).**

**Abstract:**

For this Milestone, we implemented the LCSS and the KMP algorithms. We used a python implementation for both algorithms and implemented both naive and dynamic for the LCSS and the dynamic implementation for the KMP. We then created and developed our own dataset of 7 passages, which were written and “plagiarized” minimally. We then analyzed the time output for each passage, with respect to the sum of the number of characters in the source document and the number of characters in the pattern. Finally, we break down our findings and provide some explanations for them. The link to our implementation can be found here: <https://github.com/youssefM1999/COSC320-String-Plagiarism-Project>

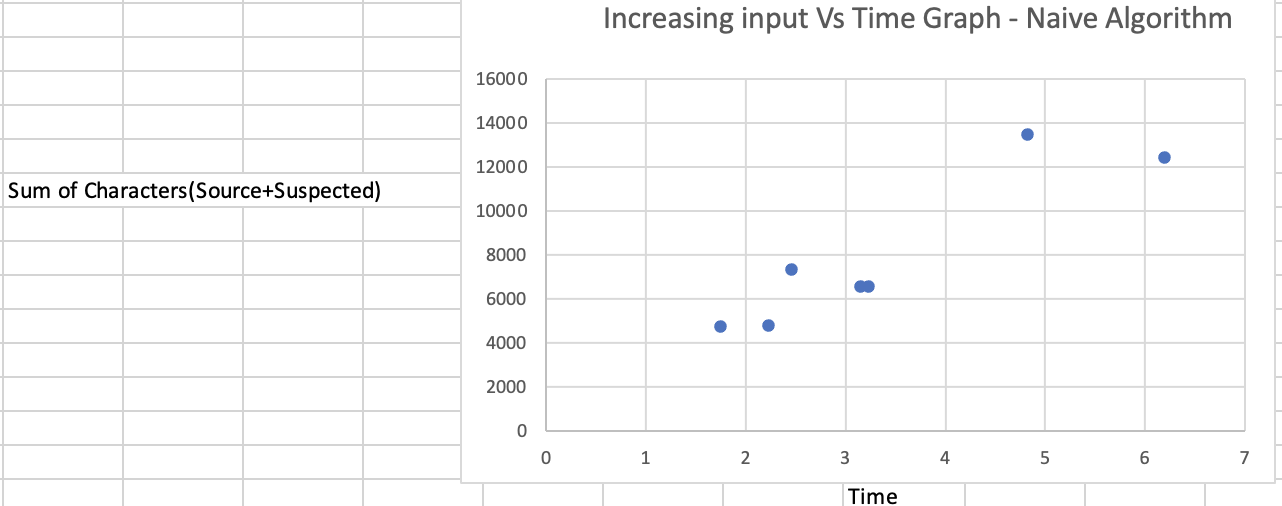
**LCSS:**

**Implementation Explanation:**

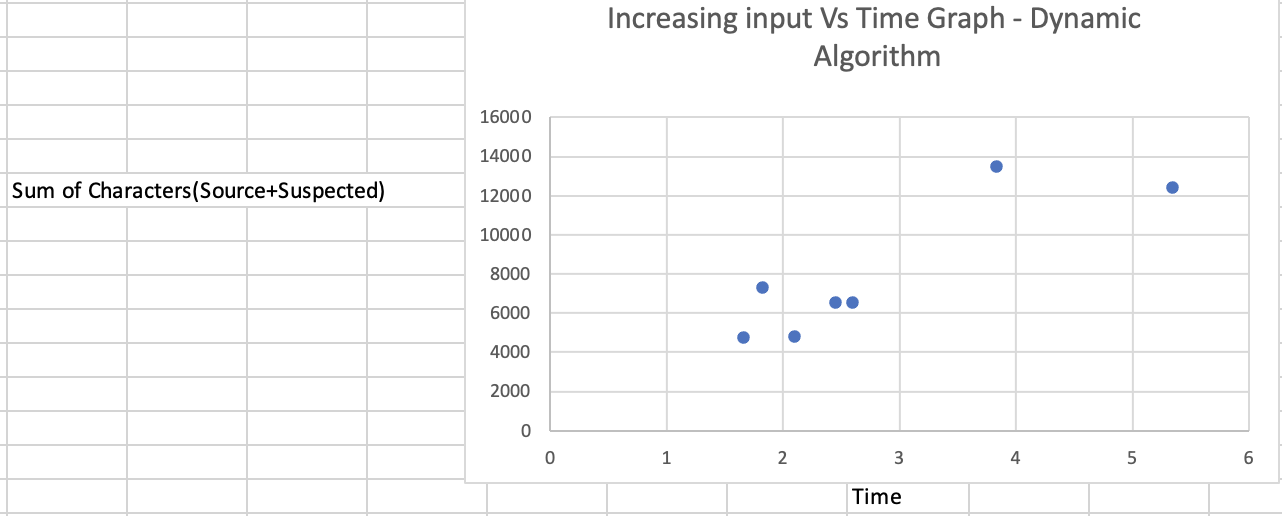
A first algorithm is a naive approach that uses a nested loop to compare every possible substring of the two strings. It starts with the first character of the first string and compares it with the first character of the second string, then it moves to the second character of the first string and compares it with the first and subsequent characters of the second string until it finds a mismatch or reaches the end of either string. The algorithm keeps track of the length of the longest common substring found so far, and updates it if a longer substring is found. The time complexity of this algorithm is O(mn^2), where m and n are the lengths of the input strings.

The second algorithm is a dynamic programming approach that uses a 2D matrix to store the lengths of the longest common substrings between all prefixes of the two strings. It starts with an empty matrix of size (m+1) x (n+1), where m and n are the lengths of the input strings. The algorithm then iterates through the matrix, comparing the characters of the two strings at each position. If the characters match, the algorithm updates the value in the matrix to be the length of the longest common substring up to that position. If the characters do not match, the value in the matrix is set to zero. The algorithm keeps track of the maximum value in the matrix, which represents the length of the longest common substring. The time complexity of this algorithm is O(mn), which is more efficient than the naive approach for large input strings.

**Figure 1.1**

****

**Figure 1.2**

****

**Conclusion:**

As shown by the graphs above, the dynamic algorithm outperforms the naive algorithm in the time taken to find the LCSS. This makes sense as the dynamic algorithm is an optimized version of the naive. The LCSS algorithm outputs the longest common subsequence, from which a threshold of plagiarism can be established. Additionally, for both algorithms, it is clear that the performance of both algorithms is dependent on the size of the file input, or the number of characters. The longer the file, the longer the time taken to find the LCSS in this case. The efficiency of the dynamic algorithm in comparison to the naive algorithm becomes most apparent as the size of the file increases.

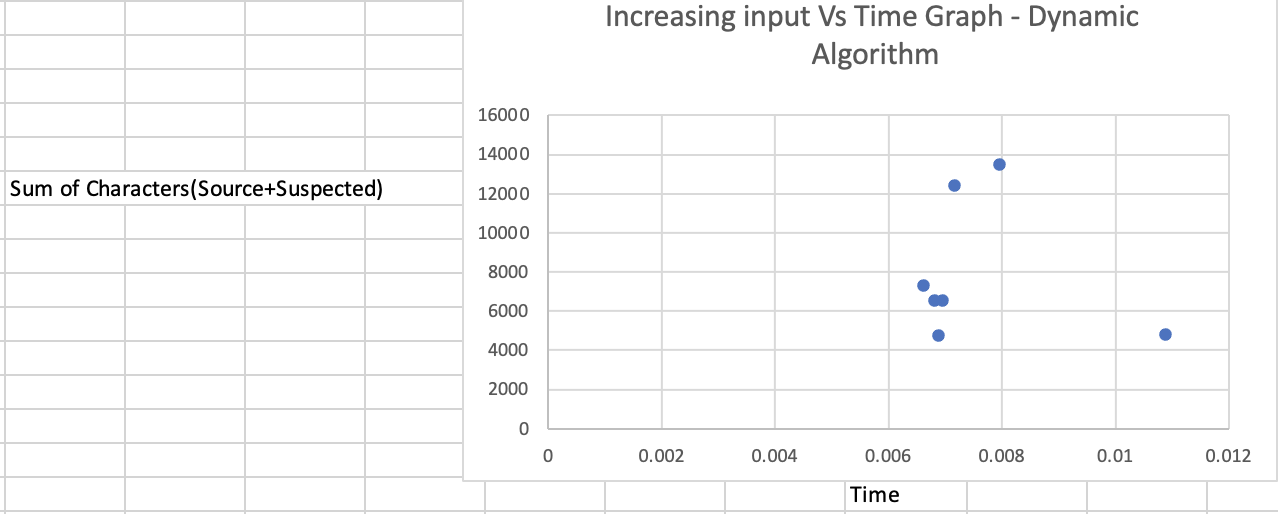
**KMP:**

**Implementation Explanation**:

`KMP()` is the main implementation of the algorithm and is a pretty straightforward Python version of the classic KMP. It has two indexes, `j` and `i`, which are used to go through the pattern and the text checking for matches. If it has a mismatch after having previously matched, it will use the `lps` array to roll the `j` indexer back a specific number of places based on the longest proper prefix which is also a suffix. It uses the auxiliary function `computeLPS()` to compute the integer array `lps`. `KMP()` outputs a list of integers that indicate where the pattern was found in the text.

`runKMP` takes to strings that represent the file to be scanned and the potential source for plagiarism. This method helps with simplifying the testing process and opens the files and scans through their content. The method processes the text into lowercase and, using a natural language tokenizer from the `nltk` package puts all the sentences in the source file into a list of strings. Each sentence in this list (`src\_sentences`) is then used as the pattern in the `KMP()` method. The method then prints every index in which plagiarism was found.

**Figure 1.3**

****

**Conclusion:**

From the graphs above, we see that the time taken by the algorithm to search for the pattern in the text is relatively constant, despite the increasing size of the input text. This is because the KMP algorithm has a time complexity of O(n+m), with n being the length of the text and m being the length of the pattern. The output shows that the time taken by the algorithm increases slightly but not significantly. This indicates that this dynamic implementation of the KMP is efficient for searching in a large bank, as it does not significantly increase as the input size grows.